Optimizing Sensor Network Reprogramming via In-situ Reconfigurable Components

Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, Frank Eliassen

Optimizing Sensor Network Reprogramming via In-situ Reconfigurable Components

AMIRHOSEIN TAHERKORDI
University of Oslo, Norway
FREDERIC LOIRET
KTH (Royal Institute of Technology), Sweden
ROMAIN ROUVOY
INRIA Lille – Nord Europe, France
and
FRANK ELIASSEN
University of Oslo, Norway

Wireless reprogramming of sensor nodes is a critical requirement in long-lived *Wireless Sensor Networks* (WSNs) for several concerns, such as fixing bugs, upgrading the operating system and applications, and adapting applications behavior according to the physical environment. In such resource-poor platforms, the ability to efficiently delimit and reconfigure the necessary portion of sensor software—instead of updating the full binary image—is of vital importance. However, most of existing approaches in this field have not been widely adopted to date due to the extensive use of WSN resources or lack of generality. In this paper, we therefore consider WSN *programming models* and *run-time reconfiguration models* as two interrelated factors and we present an integrated approach for addressing efficient reprogramming in WSNs. The middleware solution we propose, RemoWare, is characterized by mitigating the cost of post-deployment software updates on sensor nodes via the notion of *in-situ reconfigurability* and providing a component-based programming abstraction to facilitate the development of dynamic WSN applications. Our evaluation results show that RemoWare imposes a very low energy overhead in code distribution and component reconfiguration, and consumes approximately 6% of the total code memory on a TelosB sensor platform.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Distributed networks, Wireless communication*

General Terms: Design, Performance, Experimentation
Additional Key Words and Phrases: Wireless sensor networks, dynamic reprogramming, in-situ reconfigurable components, reconfiguration middleware

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are being extensively deployed today in various monitoring and control applications by enabling rapid deployments at low cost and with high flexibility. However, the nodes of a WSN are often deployed in large number and inaccessible places for long periods of time during which the sensor software, including *Operating System* (OS) and application, may need to be updated for several reasons. First, a deployed WSN may encounter sporadic faults that were not observable prior to deployment, requiring a mechanism to detect failures and to repair faulty code [Yang et al. 2007; Cao et al. 2008]. Second, in order to maintain long-lived WSN applications, we may need to remotely patch or upgrade software

deployed on sensor nodes through the wireless network. Third, the requirements from network configurations and protocols may change along the application lifespan because of the heterogeneity and distributed nature of WSN applications [Costa 2007]. Therefore, due to storage constraints, it is infeasible to proactively load all services supporting heterogeneity into nodes and hence requirement variations are basically satisfied through updating the sensor software. Finally, the increasing number of WSN deployments in context-aware environments makes *reconfiguration* and *self-adaptation* two vital capabilities, where a sensor application detects internal and external changes to the system, analyzes them, and seamlessly adapts to the new conditions by updating the software functionalities [Taherkordi et al. 2008; Ranganathan and Campbell 2003].

When a sensor network is deployed, it may be very troublesome to manually reprogram the sensor nodes because of the scale and the embedded nature of the deployment environment, in particular when sensor nodes are difficult to reach physically. Thus, the most relevant form of updating sensor software is remote multi-hop reprogramming exploiting the wireless medium and forwarding the new code over-the-air to the target nodes [Wang et al. 2006]. The *programming abstraction* and the degree of *software modularity* are two main factors that influence the efficiency of WSN reprogramming approaches. Moreover, the *run-time system* supporting reprogramming should be carefully designed in order to incur minimal additional overhead.

The early solutions in this field focused on upgrading the *full software image*. Although these provide maximum flexibility by allowing arbitrary changes to the system, they impose a significant cost by distributing wirelessly a large monolithic binary image across the network. Deluge [Hui and Culler 2004] is one of the most popular approaches in this category, offering a functionality to disseminate code updates for applications written for TinyOS [Hill et al. 2000]. Later approaches, like SOS [Han et al. 2005] and Contiki [Dunkels et al. 2006], allow *modular upgrades* at run-time with a significantly lower cost compared to the full image upgrade. SOS enables dynamic loading through position independent code which is not supported by all CPU architectures, while Contiki's solution deals with coarse-grained modules and lacks a safe dynamic memory allocation model. Finally, a few *component-based* efforts, such as FlexCup [Marrón et al. 2006], OpenCom [Coulson 2008], and Think [Fassino et al. 2002] have also been made to further customize update boundaries and maintain sensor software at a fine granularity. However, these approaches either do not satisfy all requirements of remote code update, or essentially suffer from making extensive use of memory resources.

Component-based programming provides a high-level programming abstraction by enforcing interface-based interactions between system modules and therefore avoiding any hidden interaction via direct function call, variable access, or inheritance relationships [Szyperski 2002; F. Bachmann 2000]. This abstraction instead offers the capability of black-box integration of system modules in order to simplify modification and reconfigurability of dynamic systems. Therefore, in this paper, we focus on component-based reprogramming in WSNs and reconsider Remora—a lightweight component model for static programming in WSNs [Taherkordi et al. 2010]—in order to enable compositional component reconfiguration [McKinley et al.

2004] in WSNs. The dynamicity of REMORA is achieved by the principles of *in-situ reconfigurability*, which refer to fine-grained delimitation of static and dynamic parts of sensor software at design-time in order to minimize the overhead of post-deployment updates. This also enables programmers to tune the dynamicity of the WSN software in a principled manner and decide on the reconfigurablility degree of the target software in order to tune the associated update costs. The run-time system supporting the in-situ reconfiguration of REMORA components is called REMOWARE.

The main contribution of this paper is a resource-efficient component-based reconfiguration system that is specifically designed for sensor networks and jointly considers the important aspects of a typical reconfiguration model. These aspects, including new binary update preparation, code distribution, run-time linking, dynamic memory allocation and loading, and system state retention are carefully considered by REMOWARE. To the best of our knowledge, no other earlier related work considers all these aspects in a single integrated model.

The remainder of this paper is organized as follows. In Section 2, we demonstrate a motivating application scenario that highlights the benefits of dynamic reprogramming in WSNs. After giving an overview of the contributions of this paper in Section 3, we briefly discuss the REMORA component model and its main features in Section 4. The architecture of REMOWARE reconfiguration middleware and the related design choices are presented in Section 5, while the implementation details and the evaluation results are discussed in Section 6. Finally, a survey of existing approaches and brief concluding remarks are presented in Section 7 and 8, respectively.

2. PROBLEM STATEMENT AND MOTIVATION

The success stories reported from using WSNs in industrial application areas have encouraged researchers to propose this technology in use cases dealing with every aspect of our daily life, including *home monitoring* [Wood 2006; Nehmer et al. 2006] and *healthcare* [Milenković et al. 2006] applications. The earlier solutions for proprietary home automation targeted very specific applications and operated mostly on *cable-based expensive infrastructures* [Cheng and Kunz 2009]. Moreover, these solutions addressed a small number of problems, such as satisfying security needs or the control of a limited number of devices, typically all from the same manufacturer. However, upcoming home monitoring applications are characterized as being populated with different wireless sensor nodes to observe various types of ambient context elements, such as environmental aspects (temperature, smoke and occupancy) and health conditions of inhabitants. Such information can be used to reason about the situation and interestingly react to the context through actuators.

Beyond the static software challenges concerned with developing and installing WSN-based home monitoring applications, we need to address the dynamic issues occurring during long-term monitoring with WSNs. This dynamicity is considered from two different perspectives. Firstly, such monitoring scenarios are characterized as long-lived applications, which may be deployed in a large number of dwellings. To successfully maintain (fixing bugs, patching security holes, and upgrading system and application software) this scattered infrastructure, a solution vendor needs

to establish a central updating service through which all monitored homes can be remotely maintained, instead of imposing the cost of physical maintenance to owners. When a particular update is needed, a new patch is remotely delivered to the local server in the home for further distribution of code to the target sensor nodes deployed in the home. Secondly, in home monitoring, a wide range of sensors and actuators are deployed, *e.g.*, to implement heating, ventilation and air conditioning (HVAC) control. Since different sensor nodes are likely to run different application code and interact with actuators, software reconfiguration may be needed to satisfy the dynamic requirements of owners. Finally, unlike many other WSN applications, the rate of contextual changes in home monitoring systems is basically quite high since such use cases deal with unexpected and varying human-controlled conditions [Gu et al. 2004; Huebscher and McCann 2004].

Figure 1 depicts an overview of our motivation scenario, where the platform provider not only deploys hardware and software platforms, but is also in charge of maintaining remotely the state of the application and keeping the software platform updated by sending new patches via the Internet. Each monitored home is also equipped with a gateway to receive the new code and multicast it to the target nodes inside the home. This gateway also serves as a web intermediary, enabling online home surveillance by the owner.

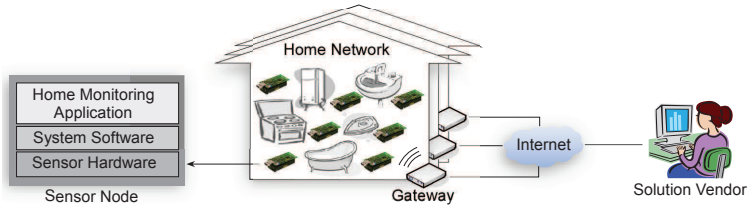

Fig. 1. Remote maintenance of WSN-based home monitoring applications.

One major problem in such distributed large-scale applications is how to remotely and individually update the software running on sensor nodes, which are mostly embedded in home appliances and may be powered by batteries with limited capacity. Rewriting the whole software image is not a feasible way to support dynamicity in such applications for the following reasons. Firstly, *preserving system state* before and after software reconfiguration is a critical requirement. In full-image updates, one of the major challenges is how to retain the values of application-wide variables when an update occurs, while in the modular update this issue is limited only to the module affected by the reconfiguration. Secondly, the size of software installed on today's sensor nodes is growing thanks to the emergence of advanced OSs for sensor platforms. These are often the dominant sensor memory consumer compared to the space occupied by application code, *e.g.*, CONTIKI as one popular OS for sensor systems consumes at least half of the code memory on a TelosB mote. Using the full-image update model, reconfiguring an application module leads to rewriting a large binary image in which the chief portion of sensor software (OS kernel and libraries) is never subject to changes. Our experimental measurements show that full-image reprogramming is much more energy-consuming than modular

update. We have evaluated this by deploying a sample monitoring application (10 KB large) on the CONTIKI OS and then measuring the reconfiguration energy cost for a given module in both mechanisms—the full-image update and the modular update. Table I shows the five main phases of a reconfiguration process, as well as the required energy for executing each phase. As shown in the table, the total updating cost of a 1 KB module in the full-image update is 19 times higher than in modular update. Although the overall cost of the full-image update is not that high considering the total amount of energy available in a typical sensor node, in use cases like home monitoring systems this cost can be a bottleneck when the rate of update requests becomes high.

Table I. A comparison between full-image update and modular update by measuring the approximate energy required to update a 1 KB module using each of these methods. The full software image is roughly 34 KB, composed of CONTIKI (24 KB) and application code (10 KB).

Operation	**Full-image Update (mJ)**	**Modular Update (mJ)**
New Update Receiving	156	4
Writing to External Flash	176	6
Linking	0	≈ 10
Reading From External Flash	31	1
Loading into Memories	19	0.2
Total	$\approx$ **380**	$\approx$ **21**

The rest of this paper addresses the main challenges to the above issues by a component-based reprogramming approach to support low cost modular updates in WSNs, and a run-time middleware abstraction providing WSN-specific core services to satisfy reprogramming needs.

3. OVERVIEW OF THE CONTRIBUTION

Figure 2 summarizes our proposal from two different perspectives. The right part outlines the programming model we offer for each particular software abstraction level. This model is a component-based approach in which every module can be developed as either a *static component* or *dynamic component*. We have already addressed the former by presenting REMORA, a resource-efficient component model for high-level programming in WSNs (REMORA v1) [Taherkordi et al. 2010]. The latter is addressed in this paper by extending REMORA with dynamic reconfiguration support (REMORA v2). In particular, REMORA v2 components can be replaced by the same component type or added/removed to/from the system at run-time.

The left part of the figure depicts a typical sensor node's software configuration with reconfiguration support based on our proposal. The reconfiguration middleware is central to our contribution and complementary to our component model. This abstraction layer consists of a set of dedicated services, which are implemented as a set of static components and support reconfiguration of REMORA v2 components. To achieve that, the middleware platform relies on low-level system functionalities providing services, such as radio and network, memory management,

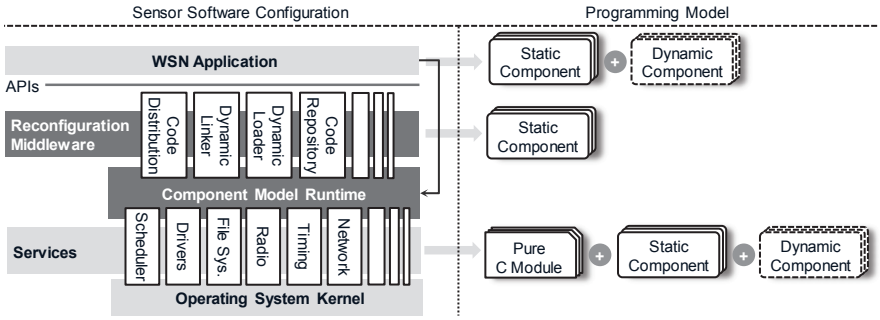

Fig. 2. Overview of the REMOWARE reconfiguration middleware.

and scheduling. For the dynamic parts of the system, the middleware exhibits an API that allows the programmer to use the reconfiguration features.

4. THE REMORA PROGRAMMING MODEL

In this section, we briefly discuss REMORA and then in the next section we reconsider the REMORA design in order to extend it with the capability of *dynamic* reprogramming and reconfiguration.

The Design Philosophy of Remora. The main motivation behind proposing REMORA is to facilitate *high-level and event-driven* programming in WSNs through a component-based abstraction. In contrast to the thought that component-based solutions are still heavyweight for current sensor platforms, special considerations have been paid in the design of REMORA to resource scarceness in WSNs, without violating its ultimate design goals. The primary design goal of REMORA is to provide a high-level abstraction allowing a wide range of embedded systems to exploit it at different software levels from *Operating Systems* (OS) to applications. REMORA achieves this goal by: *i)* deploying components within a lightweight framework executable on every system software written in the C language, and *ii)* reifying the concept of *event* as a first-class architectural element simplifying the development of event-oriented scenarios. The latter is one of the key features of REMORA since a component-based model, specialized for embedded system, is expected to fully support event-driven design at both system level and application level. Reducing software development effort is the other objective of REMORA.

A REMORA component basically contains two main artifacts: a *component description* and a *component implementation*.

Component Description. REMORA components are described in XML as an extension of the *Service Component Architecture* (SCA) model [OSOA 2007] in order to make WSN applications compliant with the state-of-the-art componentization standards. Based on the SCA Assembly Language, the component description reflects the specifications of the component including *services*, *references*, *interfaces*, *events* and *properties* of the component (cf. Figure 3). A service can expose a REMORA *interface* or a REMORA *event*. The former, described in a separate XML document, specifies the operations provided by the component, while the latter represents an event generated by the component. Similarly, a reference can request a REMORA interface, which describes the operations required by the component, or

a REMORA event, which specifies a component's interest in receiving a particular event type.

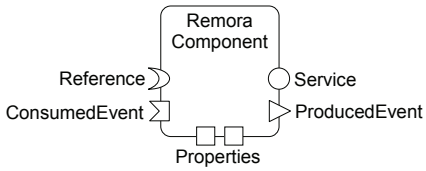

Fig. 3. A typical REMORA component.

Figure 4 shows the XML template for describing REMORA components, comprising the outer tag `componentType` and five inner tags reflecting the specification of REMORA components.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="COMPONENT_NAME”>
  <service name="SERVICE1_NAME">
    <interface.remora name="INTERFACE1_NAME"/>
  </service>
  ... other services
  <reference name="REFERENCE1_NAME">
    <interface.remora name="INTERFACE2_NAME"/>
  </reference>
  ... other references
  <property name="PROP1_NAME" type="PROP1_TYPE">PROP1_DEFAULT_VALUE</property>
  ... other properties
  <producer>
    <event.remora type="EVENT1_TYPE" name="EVENT1_NAME"/>
  </producer>
  ... other eventProducers
  <consumer operation="CONSUMER_OPERATION">
    <event.remora type="EVENT2_TYPE" name="EVENT2_NAME"/>
  </consumer>
  ... other eventConsumers
</componentType>
```

Fig. 4. The XML template for describing REMORA components.

Component Implementation. The component implementation contains three types of operations: *i)* operations implementing the component's service interfaces, *ii)* operations processing referenced events, and *iii)* private operations of the component. REMORA components are implemented using the C programming language extended with a set of new commands. This extension is essentially proposed to support the main features of REMORA, namely, component instantiation, event processing, and property manipulation. Our hope is that this enhancement will attract both embedded systems programmers and PC-based developers towards high-level programming in WSNs.

Component Instantiation. A REMORA component is either *single-instance* or *multiple-instances*. Component instantiation in REMORA is handled statically during compile time—*i.e.*, all required instances are created prior to application

startup. This relieves the run-time system from dealing with dynamic memory allocation and therefore significantly reduces overheads. Component instantiation is based on two principles: *i)* the component *code is always single-instance*, and *ii)* the component's *context* is duplicated per new instance. By component context, we mean the data structures required to store the component's properties independently from its code.

Development Process. Figure 5 illustrates the development process of REMORA-based applications. A REMORA application consists of a set of REMORA Components, containing descriptions and implementations of software modules. The REMORA compiler is a Java application, which processes the component descriptions and generates standard C code deployable as a REMORA framework. The framework is an OS-independent C module conforming to the specification of the REMORA component model. Finally, the REMORA framework is deployed on the target sensor node through the REMORA run-time, which is an OS-abstraction layer integrating the application with the system software.

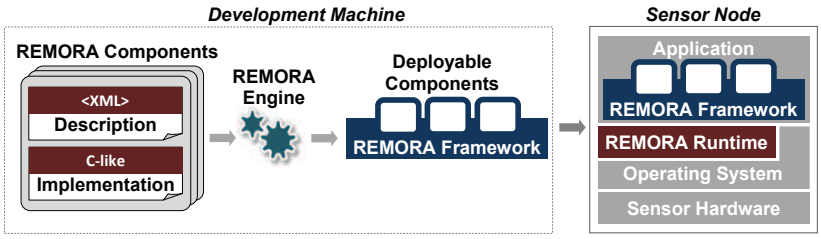

Fig. 5. Development process of REMORA-based applications.

Parameter-based Configuration. To preserve efficiency in resource usage, REMORA v1 relies on compile-time linking so that system components are linked together statically and their memory addresses are also computed at compile-time. Additionally, for multiple-instance components, all required instances are created in compiler-specified addresses prior to application startup. These constraints not only reduce the size of the final code, but also relieve the programmer from the burden of managing memory within the source code. In REMORA v1, the run-time configuration feature was also considered from a parametric perspective: A REMORA component could be configured statically by changing the behavior of its functions through its component *properties*. In particular, for the property-dependent functions of a component, the behavior of the component could easily be changed by adjusting property values and so bring a form of parameter-based configurability to the component.

5. THE REMOWARE RECONFIGURATION MIDDLEWARE

REMOWARE pays special considerations to different concerns of a standard reconfiguration model in the context of WSNs. We present below the main design choices of REMOWARE and highlight their implications on reducing the reconfiguration overhead. As *In-situ Reconfigurability* underpins most of the key features of REMOWARE, we first discuss this concept.

5.1 In-situ Reconfigurability

A fundamental challenge in enabling component-based reconfiguration in WSNs is how to efficiently provide this feature with minimal overhead in terms of resource usage. The ultimate goal of the in-situ reconfiguration model is to address this challenge. To introduce the model, we first show a close-up of a multiple-instances REMORA component in Figure 6, where the component interacts with others through services and events.

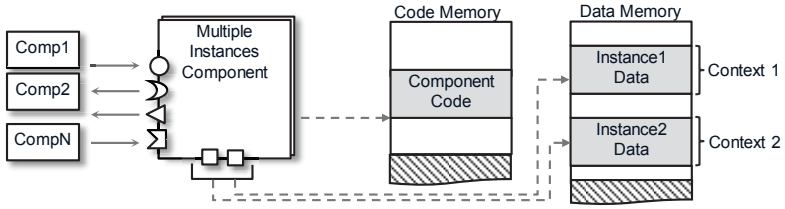

Fig. 6. A close-up of a multiple-instances REMORA component.

The main principles of the *in-situ reconfiguration model* are:

(1) The reconfiguration system updates only the component code, while the component state(s) can either be left unchanged or updated by the programmer. In particular, the reconfiguration model provides the programmer with the choice of either preserving the old component state(s) or updating it(them) with user-specified values when completing the reconfiguration;

(2) To preserve efficiency, the reconfiguration model enables the programmer to tune the overall overhead of the reconfiguration framework according to the degree of dynamicity of the system, as well as sensor platform capabilities. This is achieved by distinguishing between reconfigurable and non-reconfigurable components during software design before compiling and deploying the final sensor software. Using this strategy, the additional overhead imposed by defining an inherent static component as a dynamic component will be eliminated. We explore later in this article this overhead and discuss the impact of this strategy on the *flexibility* and *efficiency* properties of the reconfiguration framework.

(3) The scope of reconfiguration for a given component is limited to its internal design while preserving its architectural description. This constraint implies the in-situ reconfiguration model does not support interface-level reconfiguration. Hence, there is no need to manage a dynamically typed language at run-time, which is indeed heavyweight for sensor platforms.

(4) The reconfiguration model supports adding of new components and removing existing ones. In such cases, the *binding/unbinding* mechanism for loaded/unloaded components imposes only a minimal *fixed* overhead to the system, regardless of the size of the application and the number of reconfigurable components.

In REMORA v2, a component conforming to the above principles is referred to as an *in-situ reconfigurable component*. In the rest of this section, we study the

low-level design implications of REMORA v2 and address them carefully with the goal to minimize the resource overheads of reconfiguration execution. Additionally, we consider network-related issues of reconfiguration, such as packaging and distributing the new binary code of components.

5.2 Neighbor-aware Binding

Dynamic binding is one of the primary requirements of any component-based reconfigurable mechanism. A major hurdle in porting state-of-the-art dynamic component models to WSNs is their binding-support constructs, which are memory-consuming, such as MicrosoftCOM's vtable-based function pointers [MICROSOFT COM 1993], CORBA Component Model's POA [CORBA 2006], OPENCOM's binding components [Coulson 2008], and FRACTAL's binding controller [Bruneton et al. 2006]. This is due to the fact that most of those component models are essentially designed for large-scale systems with complex requests, such as interoperability, streaming connections, operation interception, nested components, and interface introspection.

In REMORA v2, we aim at reducing the memory and processing overhead of the binding model, while supporting all the basic requirements for a typical dynamic binding. To this end, we propose the concept of *neighbor-aware binding* based on the second principle of in-situ reconfigurability (cf. Section 5.1). The neighbors of a component are defined as components having *service-reference* communications or vice-versa with the component. Neighbor-aware coupling therefore refers to identifying the type of bindings between a component and its neighbors based on the reconfigurability of each side of a specific binding, as well as the service provision direction (cf. Figure 7). This is viable if the reconfiguration system can distinguish static modules from dynamic modules and the programming abstraction can provide meta-information about the interaction models between modules. These issues are jointly addressed by REMOWARE, where it relies on *component-based* updates to tackle the latter and proposes a partial-dynamic/partial-static configuration model for software modules to achieve the former.

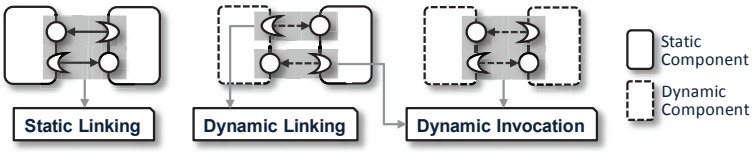

Fig. 7. Different types of neighbor-aware binding between REMORA components.

Static Linking. This form of binding refers to the function calls happening between two static (non-reconfigurable) components. According to the in-situ reconfiguration model, static components are never subject to change over the application lifespan. Thereby, we can adopt the traditional way of linking and hardcode the function invocations within each caller component in a tight fashion. In this way, static linking precludes creating metadata that is not useful at all for executing reconfiguration tasks.

Dynamic Linking. This type of linking occurs when a dynamic component invokes functions provided by a static component (*static functions*). The process of resolving such invocations is called *dynamic linking*. Whenever a new component is uploaded to the system, we need to find all the relevant invocations within the component code and update them with the correct addresses. As the addresses of static functions are fixed, one efficient way to resolve dynamic links is creating a *Dynamic Linking Table* (DLT) keeping the name and address of all static functions of static components. The immediate concern of this technique in the context of WSNs is that when we have a large number of static services, a high memory overhead may be imposed on the system. This case is somewhat addressed by the component-based nature of the system since we do not need to register the private functions of components in the DLT.

The other potential source of such memory overheads is that a typical DLT associates the function address to the *string* representation of function names and therefore depending on the size of each function name in the DLT, a significant memory overhead may be incurred. RemoWare tackles this problem by assigning an *ID* to static functions and associating the ID of functions (instead of function names) to their memory address. Hence, each row in the DLT is a <*functionId, functionAddress*> pair, needing a fixed reduced memory footprint—*i.e.*, only 4 bytes on a TelosB mote. The function ID is *automatically* generated by the part of RemoWare run-time installed on the code distribution server and the programmer therefore does not need to manually generate function IDs. Furthermore, ID is a permanent and unique value associated to a function and centrally handled by the code distribution server. Thus, a given function's IDs are identical in all nodes and the server does not require tracking such values for each node. When components are developed simultaneously by multiple teams, a set of non-overlapping ID ranges can be assigned to each development team, including third party components.

Dynamic Invocation. The physical locations of services provided by a dynamic component are not stable and change whenever the component is reconfigured. Therefore, to invoke a service function provided by a reconfigurable component (*dynamic functions*), either from a static or from a dynamic component, we need to provide an *indirect calling* mechanism. To do that, the direct invocations within the caller component should be replaced by a *delegator*, forwarding function calls to the correct service address in the memory. This delegator retains the list of dynamic service functions along with their current memory addresses in the *Dynamic Invocation Table* (DIT). Unlike the DLT, the data in the DIT is updated whenever a reconfiguration occurs, but the data structure of the DIT is designed like the DLT to overcome the memory concerns.

Decoupled Interactions. Beyond the linking types discussed above, we need to consider decoupled interactions between components, occurring in event-based relationships. Specifically, the reconfiguration system should ensure that events generated during the reconfiguration are successfully routed to function(s) of modules subscribed to those events (after completing the reconfiguration process). To this end, we need a meta-level description of events generated and consumed by modules in the system, as well as a higher-level observation on event-driven calls. In this way, the reconfiguration framework would be able to use this information

in order to fulfill event routing and so preserve the accuracy of the event management framework. We believe that "separation of concerns" in component-based models like REMORA is highly beneficial in this case since events in REMORA are described separately from components. Moreover, the REMORA engine injects the meta-information required for event routing into the component container.

Figure 8 depicts an abstract scheme of the event management model in REMORA, while the detailed description is available in [Taherkordi et al. 2010]. As a brief description, the REMORA framework periodically polls the event producer components by calling a *dispatcher* function individually implemented by all event producer components. If generating any event, the framework dispatches the generated event to the subscribed component(s) by calling a *callback handler* function within the event consumer(s). The framework accesses these functions through their pointers stored in the corresponding component state. The main concern of reconfiguring event-related components is that the physical address of their dispatcher/handler functions are likely to be changed after reconfiguration execution, while other state data remains unchanged. Therefore, REMOWARE needs to fix only the physical addresses of such functions after a reconfiguration.

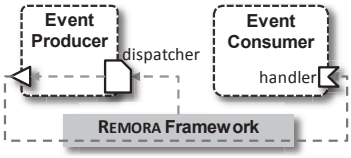

Fig. 8. Event processing scheme in REMORA.

For instance, if an event generator component is reconfigured, the reconfiguration model fixes the address of the dispatcher function used by the REMORA framework as follows:

```
for(aGenerator = listOfEventGenerators; aGenerator != NULL;
                                    aGenerator = aGenerator->next) {
  if (aGenerator->compNameId == reconfiguredCompId)
    aGenerator->dispatcher = newFuncAddr;
}
```

`newFuncAddr` is provided by the *Linker service* of REMOWARE after reading the object file of the updated component and computing the new address of the dispatcher function. Linker is one of main services provided by REMOWARE, implementing all binding models discussed above.

5.3 Component Addition and Removal

Obviously, adding new components and removing existing ones are two basic requirements to a component-based reconfiguration framework. In the previous section, we presented the neighbor-aware binding with an emphasis on component *replacement*. The other motivation behind proposing this technique is to efficiently address component addition and removal. For instance, we may decide to unload a data logger component due to the energy overhead caused by writing log data in

the external flash memory. The main concern in removing an existing component is how to handle communications referring to a removed component's services. Since a removable component should be defined as a dynamic component, its functions are dynamic and indexed in the DIT. Therefore, after removing a component we can simply map its dynamic functions to a *dummy global function* with an empty body. This function, included in the RemoWare libraries, is designed to prevent fatal errors occurring due to the absence of removed components' functions. In this way, other components dependent to an unloaded service can continue to run without exceptions.

For newly added components, the same process of dynamic linking, discussed above, can be applied, while the main issue being how to bind the new component to components that include invocations to its services. Note that prior to loading the new component, such invocations are redirected to the above dummy global function. The same solution proposed for component removal can be used for component addition, but vice-versa. When a new component is uploaded, the DIT is searched for the name of all functions of the added component. If an entry is found, the corresponding function address is corrected. Otherwise, a new entry will be inserted to the DIT with the name and address of the new function for future binding.

5.4 In-situ Program Memory Allocation

In contrast to the full software image updating model, *modular upgrade* solutions need a reliable code memory allocation model in order to avoid memory fragmentation and minimize the wasted space in the memory. Unfortunately, this issue has received little attention in the literature. Most module-based reconfiguration models either omit to consider dynamic memory allocation in sensor nodes or rely on the capabilities of OSs for dynamic memory allocation. For instance, Contiki pre-allocates only 'one' contiguous fixed-size memory space at compile time in the hope that the updated module fits this pre-allocated space. SOS [Han et al. 2005] uses simple best-fit fixed-block memory allocation with three base block sizes (16-byte, 32-byte and 128-byte blocks). Such an allocation model may waste much memory when the size of new code is slightly greater than the block size, *e.g.*, a 129-byte module in SOS needs two continuous 128-byte blocks. Lorien [Porter and Coulson 2009] creates a linked list of allocated memory spaces and uses first-fit strategy to find the first large enough gap between the allocated spaces to load a component. Long-term use of this model may lead to code memory fragmentation, in particular when the remaining free memory is limited.

We therefore propose an hybrid dynamic program memory allocation model based on the notion of *in-situ memory allocation* and *first-fit* strategy. In-situ memory allocation indicates that the updated component is tentatively flashed in its original code memory space instead of being moved to another block of memory. The immediate concern of this model is how to fit the updated component to its original memory space when it is larger than the previous version. This issue is addressed by the *pre-allocated* parameter—the extra memory that should be pre-allocated for each dynamic component. RemoWare sets a default value for this parameter (*i.e.*, 10% in the current version) applied to all reconfigurable components, while the developer can update this value for all components or a particular one. The

parameter value can be either a constant or a percentage of the actual component size, *e.g.*, by assigning 20% pre-allocation to a 200-byte component, an additional 40 bytes of code memory is reserved at the end of the component code. If an updated component cannot fit in its original memory space (including pre-allocated area), the first-fit strategy comes into play by simply scanning the free spaces list until a large enough block is found. The first-fit model is generally better than best-fit because it leads to less fragmentation. Although the in-situ allocation model is a general solution applicable to many module-based programming models in WSNs, it becomes more efficient in partially-static/partially-dynamic systems (cf. Section 5.1) since the total pre-allocated memory footprint is proportional to the number of potentially reconfigurable components, rather than all components in the system. Furthermore, in this case the pre-allocated space can be further increased in order to ensure a very large, even unlimited number of dynamic allocations.

We expect this hybrid approach to be an efficient program memory allocation model due to the following reasons. Firstly, updated components do not differ significantly from the older versions. Hence, on average there will be a minor difference only between the size of the updated one and the original component. Secondly, the developer has the ability to feed the memory allocation model with information which is specific to a particular use case and specifies a more accurate tolerance of dynamic component size.

5.5 Retention of Component State

Retaining the state of a component during reconfiguration is of high importance, *e.g.*, for a network component buffering data packets, it is necessary to retain its state before and after the reconfiguration. When considering state retention during the reconfiguration, it is very important to provide a *state definition* mechanism leading the programmer to a semantic model of global variables definition. In typical modular programming models the programmer may define global variables that are never required to be global (stateless variables). Therefore, the reconfiguration system is forced to retain all global variables (including stateless ones), resulting in additional memory overhead. In contrast, introducing the concept of state in component models like Remora prevents the programmer from defining unnecessary global variables, leading to less memory overhead when the reconfiguration system implements state retention.

When an updated version of a component is being linked to the system, RemoWare retains the previous version's state properties in the data memory and transfers them to the state properties of the updated version. This is feasible as the set of component properties never changes and the state structure of the updated component is therefore the same as previous versions. One may need to reset the value of component properties or assign a new set of values to them when reconfigured. RemoWare addresses this by calling the pre-defined onLoad function—implemented by the updated component—whenever the component is successfully reconfigured. It means that if the programmer intents to set new values to component properties after the reconfiguration, he/she must implement the onLoad function.

Figure 9 shows a simplified description of the RuleAnalyzer component in our motivation scenario, analyzing the value of home parameters, such as temperature and

light, and reasoning about the situation. This component is likely to be changed as the reasoning rules may need to be modified over the monitoring period. When the RuleAnalyzer is reconfigured, we may need to update rule parameters like threshold value for environment temperature. The following code provides an excerpt of RuleAnalyzer implementation in which we intend to increase the value of the property `tempThreshold` by 5.

```
...
void onLoad() {
  this.tempThreshold = this.tempThreshold + 5;
}
...
```

Thus, the developer can easily manipulate the value of component properties, even based on their value before the reconfiguration like the example above.

```
<componentType name="app.home.RuleAnalyzer">
  <service name="iStatus">
    <interface.remora name="app.home.IStatus"/>
  </service>
  ...
  <service name="iTemprature">
    <interface.remora name="core.peripheral.api.ITemprature"/>
  </service>
  ...
  <property name="tempThreshold" type="xsd:short"/>
  <consumer operation="onReconfigured">
    <event.remora type="core.sys.CompReconfigured" name="aRecEvnt"/>
  </consumer>
</componentType>
```

Fig. 9. Simplified description of RuleAnalyzer component in home monitoring application.

So far, we highlighted the core RemoWare services required for loading updated components on sensor nodes. In the rest of this section, we consider a complementary group of RemoWare services dealing with the construction and distribution of update code across the network.

5.6 Code Updates Management

The size of the updated code's binary image is the primary factor impacting the performance of any wireless reprogramming approach in WSNs. The binary image is an ELF file which is usually large in size and contains information that is not practically used for a dynamic loading, *e.g.*, debugging data. To reduce the size overhead of ELF in the domain of WSNs, a few extensions have been proposed in the literature like Compact ELF (CELF) [Dunkels et al. 2006] and Mini ELF (MELF) [Han et al. 2005]. In the former, the object file contains the same information as an ELF file, but represented with 8 and 16-bit data types. Although this technique can significantly reduce the size of ELF files, CELF files are still large in size and they are typically half the size of the corresponding ELF file. MELF format is used in SOS to dynamically load and unload system modules. Since the MELF

format uses position independent code and due to architectural limitations on common sensor platforms, the relative jumps can be only within a certain offset—*i.e.*, 4K in the AVR platform.

SELF Format. In RemoWare, we devise a more efficient alternative, called *Summarized ELF* (SELF), to minimize the overhead of radio communications for transmitting new updates. A SELF file contains only information required for dynamic linking of a module. In particular, SELF files include six critical segments of corresponding ELF files: code, code relocation, data, bss (*i.e.*, uninitialized variables declared at the file level as well as uninitialized local static variables), symbol table, and string table. The SELF format also removes unnecessary information in the ELF header, as well as in the header of each section. Each object file, compiled on a user machine, is first read by the *SELF Maker program* to generate the SELF format of the object file, then the newly generated SELF file is propagated through the distribution service and delivered to the target nodes.

Code Distribution. One of the primary requirements of any code updating system in WSNs is the mechanism by which the new code is distributed from a sink node, connected to a code repository machine, to other sensor nodes in a multi-hop network. Prior to formulating a relevant code distribution algorithm for a certain use-case, we need to establish a code propagation substructure that is designed specially for efficient distribution of component chunks to all or a subset of the sensor nodes. Code distribution in WSNs is a broad topic, addressed extensively in the literature [Pasztor et al. 2010; Hui and Culler 2004; Levis et al. 2004; Kulkarni and Wang 2005], and therefore we do not intend to contribute to the state-of-the-art approaches. Nevertheless, in RemoWare, we have developed a *bulk data transmission* mechanism relying on OS-specific protocols for wireless communications. This service gives to the developers the choice of identifying the target of an update, varying from a single node to the whole network.

Code Repository. Having a new version of a component uploaded on a sensor node, we may need to store a copy of the component in its external flash memory. This feature is more beneficial in adaptive applications, where the adaptation reasoning service may result in switching between different versions of a component and thus drastically reduces the cost of uploading the code update. The code repository service therefore enables the programmer to adjust the repository settings (*e.g.*, the number of versions to be stored for a component) according to the capacity of the external flash memory on a sensor node and also reconfiguration needs. This service stores the new updates by calling the operating system's library dealing with the external flash memory.

Server-based Repository. In addition to the local code repository in each node, we need to set up a repository in the code distribution server that maintains a description of the current software image of each node, representing the name and version of components currently running on each node. Having such detailed information, the programmer would be able to update sensor software with components linkable to the current configuration. In large-scale applications, the target of updates is essentially a group of nodes with the same properties (*e.g.*, a particular region of deployment). In such cases, the software image descriptor can be defined for a group in order to efficiently maintain the server repository.

Figure 10 depicts the architecture of the code distribution server. Central to the architecture is the distribution database, representing the schema of code repository data items in the server. For each sensor node (or each group of nodes) the server maintains the details of components currently deployed on the nodes (*e.g.*, name and version). Component binary images are stored in a separate table and have a one-to-many association with OS version(s) that an image can be linked to. The current OS version of each sensor node is also maintained by the server. With this data model, the Repository Manager service can track each node (or node groups) of the network and provide the required information to the server user—i.e., components currently deployed on a given node and all other components linkable to the current OS image of the node. The distribution GUI allows the server user to update the repository with the details of sensor nodes/node groups, OS versions, components and their binary images. It also communicates with the Gateway Interface service in order to distribute binary images to their destinations. FunctionID Handler is not integrated with other services of the server; rather it is used by Remora Engine to generate unique ID for component functions.

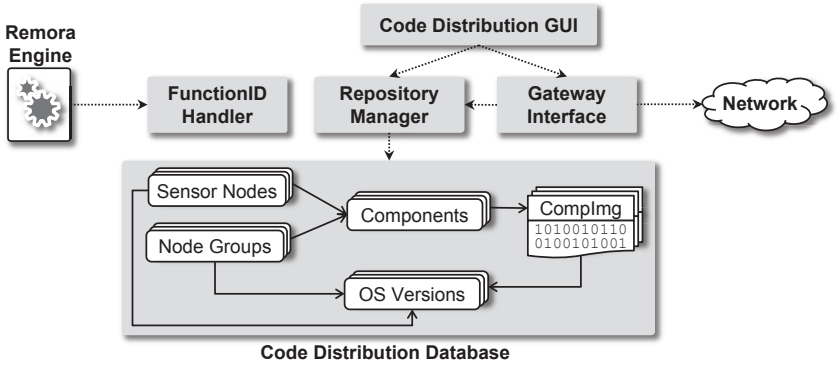

Fig. 10. The overall architecture of code repository and distribution server.

5.7 Non-functional Features

So far, we discussed the main functional features of RemoWare, while the non-functional requirements should also be carefully investigated in order to safely execute the reconfiguration needs.

Quiescent State for Reconfiguration. Reaching a quiescent state (*i.e.*, temporarily idle or not processing any request) before initiating a component reconfiguration is a critical issue in any reconfiguration mechanism. RemoWare, as any middleware framework, runs over the OS and therefore this issue is addressed depending on the process/task management model of the sensor OS. For example, the current RemoWare implementation relies on Contiki's process management system to run the reconfiguration task. Contiki processes adopt a run-to-completion policy without preemption from the scheduler. Therefore, yielding of a process is explicitly requested from the functional code and when the yielding point is reached by a process, a quiescent state is also reached for the component executed within

this process. As a result, the reconfiguration process is *atomic* in the sense that it cannot be preempted until completion. Using RemoWare on OSs that offer a preemtable task model needs a careful consideration of the safe state problem.

Reconfiguration Fault Recovery. The other crucial issue is how RemoWare handles faults occurring during the reconfiguration. Faults may happen at different steps of a reconfiguration from receiving code to writing new code in the memory, *e.g.*, inconsistencies within the SELF file uploaded to a node, errors in using the OS's libraries, and memory overflow. Writing new code into the code memory is one of most critical tasks in this context as it is very difficult to rollback to the old configuration in case of failure. RemoWare addresses this issue by storing an image of the target blocks in the external flash memory, prior to writing the new code to ROM. In case of a fault during writing the new code, RemoWare rewrites the target blocks with their original data to ROM. The high capacity of external flash memory (compared to the code memory) makes the implementation of the above technique feasible. The current RemoWare addresses only this issue, while other forms of faults are left for future work.

6. IMPLEMENTATION AND EVALUATION

In this section, we discuss the implementation of RemoWare along with the detailed evaluation results (the complete source code of RemoWare is available at [University of Oslo 2010]).

Development Technology. The core development model of RemoWare is based on the Remora component model. Remora integrates the application software with the OS through a set of off-the-shelf interfaces, wrapping OS modules and abstracting access to system services. This feature is of high importance when developing a generic middleware solution like RemoWare. Additionally, the event-driven nature of Remora, and its low memory overhead and interface-based interactions make it a suitable technology for implementing RemoWare.

Hardware and Software Platforms. We adopt Contiki as our OS platform to assess RemoWare. Contiki is being increasingly used in both academia and industrial applications in a wide range of sensor node types. Additionally, Contiki is written in the standard C language and hence Remora can be easily ported to this platform. Our hardware platform is the popular TelosB mote equipped with a 16-bit TI MSP430 MCU with $48KB$ ROM and $10KB$ RAM.

6.1 Overall Implementation Scheme

In general, RemoWare offers two sets of services for two main node types in a typical WSN. The first set is deployed on a *code propagator* node, receiving code update from a wired network and distributing it wirelessly to the target nodes in the monitored environment. The main concern of this part is adopting a widely-accepted protocol to receive new updates from common network systems. Enabling Internet connectivity in the code propagator is perhaps one of the most practical ways to connect a dynamic sensor application to the code repository server located far away from the WSNs. Furthermore, by adopting IP, sensor nodes can receive code updates even from other computing devices in a pervasive environment and thus approach to the notion of *Internet of Things* [Gershenfeld et al. 2004].

Being a REMORA application, the core interaction model of REMOWARE components is interface-based, while it uses events to listen to new updates and trigger the reconfiguration services. Figure 11 shows the configuration of REMOWARE within the code propagator node. The middleware services are exhibited by RemowareMid implementing the `IRemoWareAPI`. When RemowareMid is loaded to the system, CodeGateway listens to all new updates through the TCPListener component on a given TCP port (6510 in our current implementation). CodeGateway first stores all chunks of the component in its local file system and then distributes them to the other sensor nodes through the CodeDistributor component. This component assumes that the target nodes for distribution are specified by some other modules within the application/OS and injected to the REMOWARE through RemowareMid. Components like Network, FileSystem and BulkDataDistibutor are wrapper components that exhibit system-level functionality and also abstract REMOWARE from the OS.

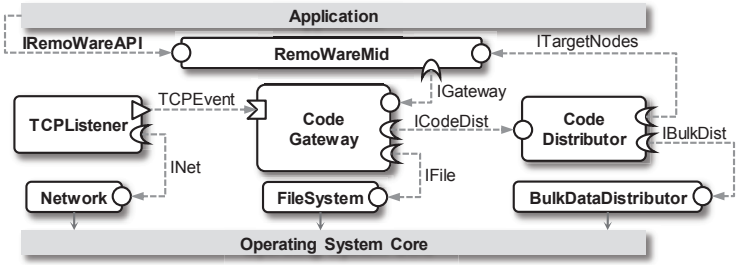

Fig. 11. REMOWARE configuration within the code propagator.

Figure 12 shows the second group of services, enabling all other sensor nodes to receive wirelessly component chunks from the code propagator and load new updates. CodeDistributor is the connection point of sensor nodes for code reception/transmission. This component is not only in charge of receiving the new code, but also able to propagate new code to other neighbor nodes. When all chunks of a component are received by CodeDistributor, it calls Linker and Loader to link the new component to the system and load it into the memory space determined by the Loader based on the in-situ strategy. For loading new components, Loader leverages the OS level component FlashMem to erase and flash the code memory.

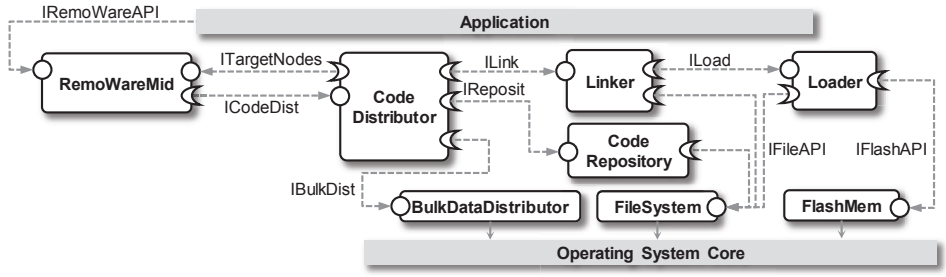

Fig. 12. REMOWARE configuration within nodes deployed in the environment.

Figure 13 shows an excerpt of the `IRemoWareAPI` interface. The `loadMiddleware` operation is central to this interface, loading and integrating the middleware services to the system when the sensor application invokes it. Although REMOWARE triggers the reconfiguration process whenever a code update is received by CodeDistributor, it also offers the `loadComponent` operation to allow the application to manually load another version of a component already stored in the external flash memory. As mentioned before, the application benefits from this operation when needing to adaptively switch between different versions of a component.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface.remora name="remoware.api.IRemowareAPI">
  <operation name="loadMiddleware" return="void"/>
  <operation name="loadComponent" return="xsd:unsignedByte">
    <in name="compName" type="xsd:string"/>
    <in name="compVersion" type="xsd:unsignedByte"/>
  </operation>
  <operation name="unloadComponent" return="xsd:unsignedByte">
    <in name="compName" type="xsd:string"/>
  </operation>
  <operation name="getCurrentVersion" return="xsd:unsignedByte">
    <in name="compName" type="xsd:string"/>
  </operation>
  ...
</interface.remora>
```

Fig. 13. An excerpt of the `IRemoWareAPI` interface.

In the rest of this section, we discuss the key techniques used to implement REMOWARE services and we report the evaluation results.

6.2 New Code Packaging

The new updates are prepared according to the SELF format, containing code, data, code relocation, data relocation, and symbol table. We have developed a C program (SELFMaker) that processes the ELF file on the user machine and generates the equivalent SELF file. In the following, we measure the overhead of SELF file distribution.

Evaluation Results. To evaluate the SELF format, we measure the size of the SELF file for four different components and compare them with the size of equivalent CELF files. The results are listed in Table II and show that the SELF format brings a fixed small improvement over the CELF format, where SELF is at least 235 bytes smaller than the equivalent CELF file. This optimization is chiefly achieved thanks to removing unnecessary information from section headers in the SELF model. The SELF format reduces the size of main ELF header from 52 to 28 and the size of each section header from 40 to 16. Given that both SELF and CELF contain seven sections, a SELF object file is at least 192 bytes smaller than the equivalent CELF file. Table II also shows the estimated energy cost of distributing components to other nodes through the CC2420 radio transceiver of TELOSB based on CONTIKI's RIME protocol. We quantify the energy consumption by using CONTIKI's software-based online energy estimation model [Dunkels et al. 2007]. The results for different file sizes show that both SELF and CELF impose a negligible

energy overhead, even for large files. The comparison of energy usage of SELF and CELF files indicates that the overhead of both formats are at the same order. In fact, the efficiency of SELF is essentially in memory usage on sensor nodes. Compared to CELF, the SELF format can save at least 235 bytes of memory for each component image, thereby, the Repository service is able to store a larger number of components in the external flash memory.

Table II. The overhead of the CELF and SELF file formats in terms of bytes and estimated reception energy for four REMORA components.

Component	Code size (bytes)	Data size (bytes)	CELF size (bytes)	SELF size (bytes)	CELF size overhead (bytes)	CELF reception enegy (mJ)	SELF reception energy (mJ)
Leds	140	8	1139	904	235	4.03	2.87
Blinker	212	1	1592	1356	236	7.14	6.11
Router	362	6	2484	2248	236	12.22	9.70
RuleAnalyzer	636	32	3064	2715	349	13.80	12.38

6.3 Code Repository

The implementation of the Repository service relies on the CONTIKI file system (COFFEE) [Tsiftes et al. 2009] to store and retrieve the SELF files. The performance of Repository is considered from two viewpoints. First, the size of the binary object of the updated component directly impacts the overhead in using the external flash memory. We minimize this overhead in REMOWARE by proposing the SELF format to allow an increased number of components to be archived in the external memory. Secondly, the OS's file system imposes its own additional memory and processing overhead to the Repository service for storing SELF images.

Evaluation Results. The size of component chunks is the first factor affecting the performance of the Repository service. As the size of a chunk is reduced, a higher number of network packets must be transmitted for a component and thus more writing tasks are performed by Repository. Figure 14 shows our experimental results of storing a Leds component (904 bytes) on the external flash memory with different chunk sizes. We repeated our experiment 15 times for every chunk size. As shown in the graph, the estimated energy cost for 144-byte chunks is 6.59 mJ, while this overhead is increased to 42 mJ for 24-byte chunks. As a result, the protocol for bulk data transmission directly influences the efficiency of Repository. The general observation is that writing in the external memory is costly and therefore Repository may become a performance bottleneck and impose a significant overhead to the system. One solution to reduce this overhead is to store different versions of a reconfigurable component in sensor nodes before deployment.

Repository is not only used during receiving or distributing new code, but also used by Linker to read and update symbol values in the SELF file. Figure 15 plots the estimated energy requirements for resolving *i)* dynamic links within a component, and *ii)* dynamic invocations, *e.g.*, for a component with 18 dynamic links, the overhead is 15 mJ. Since for dynamic invocations, REMOWARE updates only the

DIT in RAM, we have only the overhead of reading from the SELF file, while for the dynamic links we need to update the SELF file and therefore the consumed energy is higher.

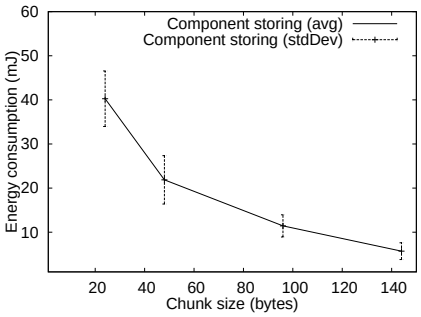

Fig. 14. Energy cost for storing the Leds component (904 bytes) on the external memory with respect to the chunk size.

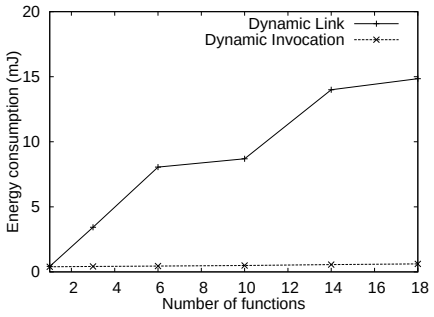

Fig. 15. Repository-related energy cost for resolving dynamic links and dynamic invocations within a component.

6.4 Component Linking

Linker reads the SELF file of a component through the Repository and then starts the linking process. Figure 16 depicts a simplified model of dynamic linking in which a reconfigurable component contains two static function calls to two different static components. The physical addresses of all static functions are also listed in the DLT.

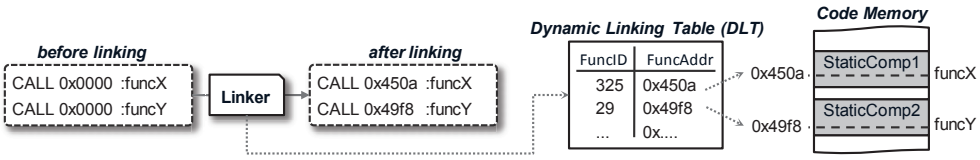

Fig. 16. Binary code of a reconfigurable component before and after dynamic linking.

The first linking phase is allocated to resolving *dynamic links* (cf. Section 5.2). To do that, Linker first extracts all static function calls (to the other components) from the corresponding symbol table in the SELF file, then it searches in the DLT for their physical address. The DLT is also filled by the REMORA engine during application compilation as illustrated by the following code:

```
const struct StaticFuncAdr staticFuncAdr[...] = {
  {2,  (void *)&_002_ISystemAPI_currentTime},
  {4,  (void *)&_004_ISystemAPI_activateSensorButton},
  {12, (void *)&_012_IFileAPI_fseek}, ...
}
```

This technique arises an important question: for resolving a dynamic link how can Linker find the ID of corresponding static function? The REMORA engine addresses this issue by prefixing the ID of functions to their names before compilation. In this way, Linker is able to extract the ID of functions by parsing function names available in the symbol table. When the correct address of a static function is found, Linker updates the corresponding static calls within the code section of the SELF file and so fulfills the first linking step.

The next step is dedicated to replacing dynamic calls within the user code by a delegator function call (cf. Figure 17), *e.g.*, `iLeds.onLeds(LEDS_RED)` is a user code calling a function of `iLed` service, while the REMORA engine replaces it by `(delegate(3))(LEDS_RED)` in which 3 is an index for the `onLeds` function. We offer an *index* for each dynamic function to minimize the overhead of searching the DIT for each dynamic call. This way, Linker can directly fetch the address of functions without searching the DIT. When the generated code is compiled and linked to the system, all calls to a dynamic component are handled through the delegator module. Maintaining the DIT is the other task of Linker in this phase. Linker extracts the name and address of all *service functions* from the SELF file and updates the DIT with the new addresses. The new address of functions is also computed according to the memory address of the component, obtained based on the in-situ memory allocation mechanism.

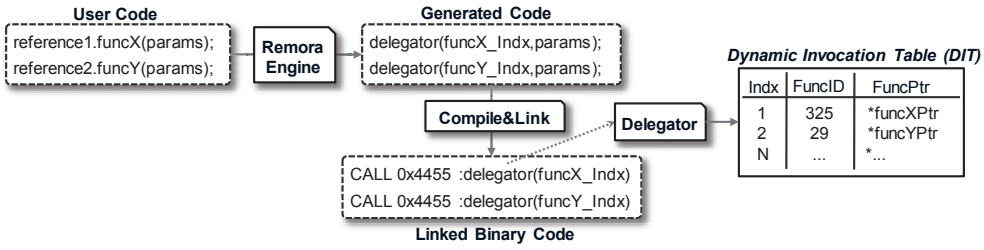

Fig. 17. Steps required to resolve invocations to reconfigurable components.

Component state retention is also implemented by Linker. As mentioned before, the component state represents the current value of component properties. The high-level definition of component properties (by the programmer) is aggregated into a C struct (*state struct*) by the REMORA engine and stored in the data memory of the sensor node. The starting memory address of the state struct will also be added to the metadata of the component by Linker. When linking an updated image of a component, Linker relocates the memory address of the new state struct to the old one using the above metadata. Additionally, Linker looks for the onLoad function in the symbol table. If found, it calls this function when the component is successfully loaded to the memory. This allows the programmer to change the component state with his/her own set of values upon component reconfiguration.

Evaluation Results. We consider two different *worst-case* configurations of components to evaluate dynamic linking and dynamic invocation. To assess the former, we focus on the DLT as a potential bottleneck and configure our system with a DLT

with *i)* 400 entries and *ii)* 650 entries, meaning that all application components together exhibit 400/650 static functions. We believe that in a real WSN application the number of service functions is far less than these values. We measured the energy requirement of linking a component, containing 3, 4, 6, and 8 static function calls, against the two DLTs above. Figure 18 shows the energy overhead of two main dynamic linking tasks: searching in the DLT and updating the SELF file of the component afterwards. Our observations indicate that the size of the DLT is not a concern, while the main overhead is incurred from updating the SELF file in the external memory. In the second configuration, we compare the overhead of dynamic invocation and the normal way of direct invocation (Figure 19). The bottom line in the graph shows the overhead of dynamic invocation, reaching 0.02 s for 5000 invocations of a dynamic function. We consider this overhead to be very low and believe it to be negligible.

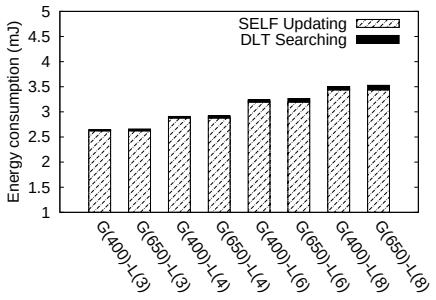

Fig. 18. The energy requirement of dynamic linking. G(x): number of global static functions, L(x): number of static function calls within the component.

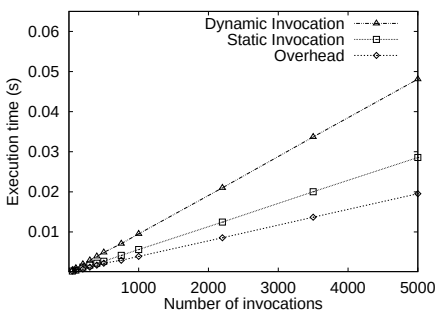

Fig. 19. Comparison between the CPU overhead of dynamic invocation and equivalent static invocation.

In addition to the above performance metrics, it should be noted that greater efficiency is achieved by being able to distinguish static components from dynamic components. Without this feature, linking tables should maintain a large number of functions (including functions belonging to OS modules), while many of those functions may never be used. For example, CONTIKI's symbol table requires at least 4 KB of code memory [Dunkels et al. 2006] to keep track of the names and addresses of system functions.

6.5 Component Loading

The *Loader service* is in charge of implementing the in-situ memory allocation model and loading the new component in the allocated space. Before doing that, this service performs the crucial task of *relocation*. Relocation refers to the process of resolving the memory addresses of locally-defined symbols based on the starting address of the updated component in the memory. The loader first allocates memory to the updated component, reads the binary object of the component from *Repository*, performs relocation, and finally writes the component code into memory. The only metadata required to do loading is the memory-related information

of dynamic components, such as the component's code memory address, code size and the component's data memory address (for doing relocation). This information is acquired in two steps during sensor application compilation.

Most of the sensor hardware platforms allow only block-wise update on ROM—*i.e.*, 512 bytes in the TelosB mote. To ensure a more fine-grained update, Loader copies the binary code of other components within the target block to RAM. Afterwards, it erases the target block and writes the whole block—including other components' code and the new component's code—to the memory. If the starting and ending address of the component code are in two different blocks, Loader takes care of two adjacent blocks.

Evaluation Results. We consider two forms of memory boundaries for the Leds component (see above) to evaluate Loader: *i)* the whole component code is located in one block of memory, and *ii)* the component code is divided in two adjacent blocks of memory. Table III lists the energy overhead of the main tasks of Loader for loading a new Leds component. The results show that *block update* is the main resource consumer, although the total overhead of component loading is very low and negligible, even in the case where two blocks are involved.

Table III. Energy overhead of loading the Leds component.

Component Memory Boundaries	Linked Comp. Reading (mJ)	Block Update (mJ)	Other Loading Logic (mJ)	Total (mJ)
One block	0.015	0.137	0.013	**0.16**
Two blocks	0.015	0.274	0.021	**0.31**

The other major concern is the performance of the in-situ memory allocation model. To assess this, we developed a program simulating the in-situ memory allocation model for the TelosB mote. To accurately analyze the behavior of the memory allocation model, we assume that the whole programmable space in the code memory (~47 KB out of 48 KB) is subject to dynamic allocation. The metric for performance measurement is the total amount of dynamic memory allocations that can be accomplished before ROM becomes fragmented. The main factors affecting this metric include: the size of the deployed software code (S_{soft}), the percentage of the components that are dynamic (P_{dyna}), the average size of component code (S_{comp}), and size of the pre-allocated memory (S_{pre}).

In the first experiment, we investigate the impact of pre-allocated memory space on the maximum number of allocations. The default value of the above factors are chosen based on an average setting in which $S_{soft} = 40KB$ (including both OS and the application), $P_{dyna} = 50\%$, and $S_{comp} = 300$ bytes (according to our measurement for a typical Contiki-based software). Figure 20 shows the behavior of the in-situ model when the value of S_{pre} ranges from 0% to 20% of component size. The new size of the component (after the reconfiguration) is the main factor in this scenario, *e.g.*, when the new component size ranges from -5% to 20% of its original size, with 15% pre-allocation, the in-situ model can continue memory allocation up to 300 times (in average). In the same range, when the pre-allocation size approaches the upper range limit (20%), a high number of allocations is achieved,

e.g., 18% pre-allocation allows approximately 500 new allocations. This is because the size of the most updated components fluctuates within the pre-allocated space. However, as S_{pre} reaches 5%, we observe a small reduction in the maximum number of allocations. The reason is that this value is close to a turning-point in which about half of the updated components cannot fit in the allocated spaces and at the same time the total pre-allocated space (equal to $S_{soft} \times P_{dyna} \times S_{comp} \times S_{pre}$) is significant. Although for smaller values (*e.g.*, 2%) the rate of failed allocations is higher, less memory space will be pre-allocated and consequently more free space is available for the first-fit allocation. When $S_{pre} = 0$, the in-situ model acts as a first-fit allocator. In the other ranges, 5% to 25% and 10% to 30%, we face a smaller number of successful allocations and the in-situ model acts like the first-fit model. The main conclusion from the above experiments is that an accurate estimation of the pre-allocated space can drastically increase the performance of the memory allocation, while, in the worst cases, it behaves like the first-fit model without significant performance degradation.

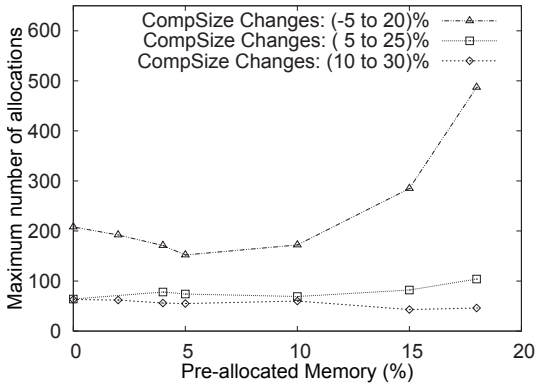

Fig. 20. Behavior of the in-situ memory allocation model.

The second experiment is to compare the in-situ model with the first-fit strategy based on the following assumptions: *i)* 47 KB of program memory is available for allocation and *ii)* the updated component's size fluctuates between -5% and 20% of its original size. We consider three different sizes for the deployed software: 38 KB, 40 KB and 42 KB. Additionally, a set of modest values are assigned to the in-situ parameters: $P_{dyna} = 50\%$ and $S_{pre} = 10\%$. Figure 21 shows the maximum number of allocations allowed by each mechanism when the average size of components varies from 200 bytes to 500 bytes (error bars show standard deviations). Whereas both mechanisms allow unlimited number of allocations when the size of code is smaller than 38 KB, for larger applications the in-situ model enables at least twice the number of allocations as compared to the first-fit model (for medium-size components, *i.e.*, 300 bytes). In conclusion, our observations show that the in-situ strategy can improve significantly the performance of memory allocation when the remaining free memory space on the ROM is not high. This case is so likely to

happen in the typical sensor software given that a large memory space is usually occupied by OS.

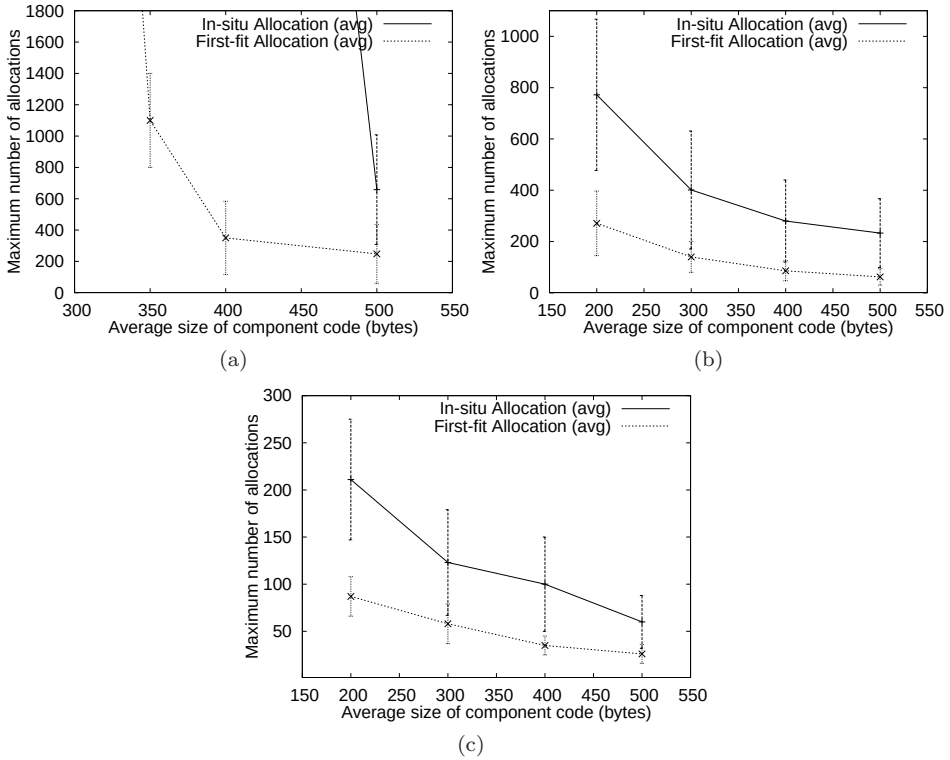

Fig. 21. Memory allocation increase vs. average component size given that the size of deployed software is: a) 38 KB, b) 40 KB, and c) 42 KB.

6.6 Putting It All Together

In this part, we consider the overall performance of RemoWare. The results reported in this section will be also compared to related work in Section 7. As the middleware platform runs over Remora run-time system, we also need to take into account the indirect overhead of the Remora run-time imposed on the middleware framework. We organize the overall evaluation according to the two dominant performance figures of WSNs: memory footprint and energy consumption.

Memory Footprint. High memory overhead is one of the main reasons behind the unsuccessfulness of high-level software frameworks in WSNs. In RemoWare, we have made a great effort to maintain memory costs as low as possible by proposing in-situ reconfigurability and minimizing the metadata required to handle dynamic reconfiguration.

The memory footprint in RemoWare is categorized into a *minimum overhead* and a *dynamic overhead*. The former is paid once and for all, regardless of the

amount of memory needed for the dynamic components, while the latter depends on the number of reconfigurable components and their services. The minimum overhead also includes the memory footprint of the REMORA run-time. Table IV shows the minimum memory requirements of REMOWARE, which turn out to be quite reasonable with respect to both code and data memory.

Table IV. Minimum memory requirement of REMOWARE.

Module	Code Memory (bytes)	Data Memory (bytes)
REMORA run-time	494	14
Distributor	300	64
Linker	1228	20
Loader	642	0
Repository	122	0
Other Helper Comp.	206	0
Total	**2992**	**98**

As CONTIKI consumes roughly 24 KB (without µIP support) of both these memories, REMOWARE occupies approximately 6% of the total ROM and thus it has a very low memory overhead considering the provided facilities and the remaining space in the memory. Table V reports the dynamic memory overhead of REMOWARE on the TELOSB mote with a CPU based on 16 bit RISC architecture. In particular, using function ID, instead of function name, significantly reduces the dynamic overhead so that for each static/dynamic function we only need 4 additional bytes of ROM/RAM. Dynamic components also have a fixed overhead of 8 bytes, which are the metadata representing the memory map of component.

Table V. Dynamic memory overhead of REMOWARE.

Entity	Code Memory (bytes)	Data Memory (bytes)
Static Function	4	0
Dynamic Function	0	4
Dynamic Component	8	0

The memory requirements of dynamic invocations are not limited to the above *direct* amounts, when they are invoked by other components. As discussed in Section 6.4, all dynamic calls within the code are replaced by equivalent delegator function calls. The minimum additional memory requirement of such calls is 6 bytes and this amount is increased by 2 bytes for each function parameter, *e.g.*, `(delegate(3))(LEDS_RED)` needs 8 more bytes of ROM than `iLeds.onLeds(LEDS_RED)` when compiled. Thus, depending on the number of calls to a dynamic function the *indirect* memory overhead will be increased.

Total Energy Overhead. Finally, we consider the total energy requirements for reconfiguring the components we mentioned at the beginning of this section. Table VI shows the results of our measurements for two main phases of reconfiguration: *component distribution* and *component reconfiguration*. For each component, the number of dynamic links (carried out by Linker) is also specified as it is a dominant part of the total consumed energy. For example, in the case of the Router component, the high number of dynamic links causes a moderate increase in required time for the reconfiguration, in addition to the overhead induced by its size. We believe that the total overhead of RemoWare reconfiguration model is acceptable with respect to the high-level features it provides to the end-user, even though this overhead can be significantly reduced by improving the implementation of the Repository service.

Table VI. Total time overhead of reconfiguring four components with different sizes.

Component	SELF Size (bytes)	Dynamic Links	Distribution Time (ms)	Reconfiguration Time (ms)	Total (ms)
Leds	904	1	2.17	18.99	21.16
Blinker	1356	5	4.63	22.59	27.22
Router	2248	12	7.35	45.54	52.89
RuleAnalyzer	2715	4	9.36	50.03	59.39

6.7 Reinvestigation of Results

Beyond the above in-dept evaluation of RemoWare, we need to reinvestigate the results that are of importance and play a crucial role in the design and implementation of our reconfiguration framework.

The second in-situ reconfiguration principle—the ability to distinguish between the static and dynamic components at design time—is one such result. This principle incurs a tradeoff between *optimization* and *flexibility*. Our evaluation shows that marking a component as a static module can optimize the performance of the middleware by: *i)* avoiding the metadata required to retain a component's memory information and its interfaces with others, and *ii)* eliminating the in-situ dynamic memory allocation cost. We also illustrated that the above overheads are not significant when the remaining code memory capacity on the node is not very low. On the other hand, the programmer can achieve a higher level of flexibility by identifying all components as reconfigurable modules. However, this can degrade the performance in the order of number of all components. As a result, this principle is proposed to leave some optimization and flexibility decisions to the programmer. In this way, she/he can tune the above factors according to the target application's attributes and the resource-richness of sensor nodes.

The above issue can be further explored by evaluating the *processing* and *memory* overheads when all software components in the system are dynamic. The processing cost of dynamic calls has already been considered in Section 6.4. In Figure 19, we have shown that the additional CPU overhead of dynamic invocations is negligible even when the number of dynamic invocations reaches 5000. According to the

measured execution time, the average overhead for one dynamic call is less than 4 microseconds compared to the equivalent static call. Obviously, this overhead is very small, even for long-lived applications. Thus, our approach does not impose significant processing cost for a completely dynamic system.

To precisely evaluate the memory overhead, we have expanded dynamicity to the whole system and componentized a number of OS modules using the REMORA component model in order to measure the memory overhead when all those components are reconfigurable. Table VII shows the results for three system components dealing with file management (FileManager), sensor LEDs (Leds), and networking (PacketQueueBuffer). Let us focus on FileManager as an example to illustrate the indirect memory cost it imposes to the system due to being a dynamic component. This component exposes 9 interface functions. According to Table V, it consumes $8 + 9 \times 4 = 44$ bytes direct code memory. These functions are invoked in 39 different places of code, including within the REMOWARE components and within the sample home monitoring application's components. The average number of parameters for each function is approximately 3. Therefore, the total indirect memory usage becomes $39 \times 6 + 39 \times 3 \times 2 = 468$ bytes. Similarly, the memory cost for the other two components is calculated and the total cost for these three components becomes roughly 1.6 KB. It should be noted that the overhead is considered only for three OS components. This can be drastically increased when all components in the system are dynamic. Therefore, selectable staticness/dynamicness can be a very important feature, especially in cases where system-level components can also be dynamic.

Table VII. Indirect program memory requirements for three different OS modules when developed as dynamic REMORA components.

Component	Number of Calls	Average Number of Parameters	Cost from Table V (bytes)	Total Cost of Dynamic Calls (bytes)	Total (bytes)
FileManager	39	3	44	468	512
Leds	25	1	32	200	232
PacketQueueBuffer	72	1	32	576	608

The other observation in this study is that the size of a component is not always the main factor influencing the memory overhead, but the "number" of calls to the component's functions may be more critical. In the above experiment we have selected components in three different ranges: large (FileManager), medium (Leds), and small (PacketQueueBuffer). As shown in Table VII, the cost of FileManager itself is 44 bytes (as a large component), while the total indirect cost of dynamic calls to this component's functions is 468 bytes. Furthermore, PacketQueueBuffer is a small-size component, but the indirect total cost of dynamic calls for this component (576 bytes) is quite higher than other two larger components.

One may argue that delimiting the reconfigurable portions of software at design time is not always useful as one main reason to change the deployed software is to handle unforeseeable problems, *e.g.*, fixing unanticipated bugs. Although we

believe this makes sense for many system-level components, there are still many other components that merely act as utility modules with very basic functions, *e.g.*, the Leds component. As such components are general-purpose and frequently used by other components in the system, they may induce a significant memory overhead when identified as dynamic components.

The dynamic memory allocation model is the other concern that may affect the efficiency and applicability of RemoWare, although we have shown that for a typical application this model ensures a very large number of memory allocations. The hybrid approach is essentially proposed to ensure the optimal usage of free memory spaces. As mentioned before, the programmer can control the memory allocation mechanism by regulating the value of the pre-allocation parameter. Therefore, by a proper estimation of this value the allocation process becomes a simple non-hybrid model such that the updated component's code fits in its own memory space and is not relocated to some other address in the memory computed by the first-fit strategy. In this case, RemoWare can keep allocating memory to updated components unlimitedly over the application lifespan.

6.8 RemoWare Beyond the State-of-the-art?

In order to support the claim that RemoWare is the first to provide an efficient, practical component-based software update model for WSNs, we compare it with other existing component-based approaches in this area with respect to different metrics such as usability on various OS platforms, memory requirements, and memory allocation models. Section 7 further discusses the other shortcomings of those approaches. Table VIII shows a summarized comparison of RemoWare with other works proposed in this category. The first metric is concerned with the usability of existing models on different OSs. Whereas most of existing approaches are dependent on a particular OS, RemoWare can be exploited on any system software written in the C language. The next three columns in the table show the fixed and dynamic memory costs. Compared to RemoWare, all other models come with high memory demand, except FiGaRo which is limited to Contiki's capabilities for dynamic reconfiguration. The memory allocation model is perhaps the most important innovation of RemoWare as it is not addressed in most other proposed systems. Think can be adapted to reuse existing memory allocation models, while FiGaRo relies on Contiki's solution—'one' pre-allocated contiguous memory space. Finally, all existing models use their own proprietary object file format. As illustrated in Section 6.2, SELF has further reduced the file size compared to CELF—one of the most compressed and summarized formats.

In addition to the component-based solutions, we compare RemoWare with other approaches sharing the core contributions of RemoWare, such as modular reconfiguration models for WSNs. SOS [Han et al. 2005] is perhaps the most relevant approach enabling modular update in WSNs by installing a static OS kernel on all the nodes and supporting dynamic fine-grained deployment of other higher-level modules. The main drawback of SOS is that it uses position independent code to link and load updated modules. Unfortunately, position independent code is not available for all sensor platforms and this limits the applicability of SOS. Nevertheless, SOS's solution for dynamic interactions between modules is analogous to RemoWare's binding model. Interactions between dynamic modules in SOS oc-

Table VIII. Overview of existing component-based approaches to WSN reconfiguration.

Approach	OS Platform	Core Size (KB)	Cost per Comp. (Bytes)	Cost per Service (Bytes)	Memory Allocation Model	Update Image Format
FlexCup	TinyOS	16	0	0	Not needed	binary comp.
Lorien(OpenCom)	Lorien	5.5		350	First-fit	ELF
Think	OS-Indep.	2	102	16	Adaptable	binary comp.
FiGaRo	Contiki	2	15	18	Contiki	CELF
LooCI	SunSpot	20	587	N/A	N/A	Java bytecode
Remora&RemoWare	OS-Indep.	2.9	8	4	In-situ	SELF

cur via indirect function call (*inter-module calls*). The same approach with minor difference is used by dynamic modules to call static functions within the kernel (*kernel calls*).

Being a component-based reconfiguration model, RemoWare offers essential improvements over SOS's dynamicity. Firstly, SOS's reconfiguration model relies on kernel features, thereby kernel upgrades require replacement of the entire kernel image which is not the case in RemoWare. This is because RemoWare's key reconfiguration tasks are implemented independently of the underlying system software, *e.g.*, the in-situ memory management model, the Remora framework, and the reconfiguration engine as a whole (cf. Figure 11). Although services like FileSystem may rely on OS features, typical system software is assumed to expose such a service for sensor nodes with external flash memory. Secondly, the meta-information about component interactions makes it possible to automatically and efficiently register dynamic functions with the RemoWare run-time, while registration of both dynamic and kernel functions in SOS must be carried out manually by the programmer. Thirdly, it is not clear how SOS deals with event-driven interactions between modules when undertaking the reconfiguration. As discussed before, RemoWare as a component-based update solution precisely addresses this issue using meta-information about event-driven invocations provided by the Remora framework.

Additionally, kernel calls in SOS turns out to have the same order of overhead as inter-module calls. To clarify this issue, we investigated the memory and processing overheads of different invocation types in SOS and compare them with RemoWare. As shown in Table IX, the processing overhead for inter-module calls in SOS is 17 clock cycles (compared to direct static call), while RemoWare has reduced it to 10 clock cycles. This improvement is achieved as RemoWare adopts a more simple indexing model for locating the addresses of dynamic functions. The memory consumption of inter-module calls in SOS is similar to RemoWare discussed in Section 6.6 and Section 6.7, but with additional 4 bytes overhead in each call. Table IX also shows that kernel calls come with overheads similar to inter-module calls, while RemoWare can mitigate this overhead by differentiating between static and dynamic kernel components.

As a result, in real applications with numerous calls to kernel/dynamic modules, SOS may impose a significant memory cost as the above overheads are fixed and not amenable by the programmer. However, SOS can be a right and efficient choice

Table IX. A comparison between the processing and memory overheads of SOS and RemoWare for different calling types.

System	Inter-module Call		Kernel Call	
	Clock Cycles Overhead	Memory Cost (byte)	Clock Cycles Overhead	Memory Cost (bytes)
SOS	17	10+	8	6+
RemoWare	10	6+	0 or 10	0 or 6+

for configurations in which both kernel and application are small in size and the number of interactions between modules is not high.

7. RELATED WORK

There have been several proposals for enabling reprogramming and providing software reconfiguration in sensor networks. We classify them into four main categories and discuss related works in each category.

Full Image Reprogramming. Deluge [Hui and Culler 2004] is perhaps the most popular work in this category, supporting the replacement of the full binary image of TinyOS [Hill et al. 2000] applications through a networked bootloader and distribution protocol. Due to the large size of the monolithic binary image, Deluge incurs a high energy overhead for code transmission. Reinstalling the full application also disrupts the running application, resulting in a loss of recent data and resources. To overcome the first problem, approaches like [Jeong and Culler 2004; Reijers and Langendoen 2003] compare the new code image with the previously installed one and transmit only differences. The main drawback of such approaches is that algorithms for detecting differences are complex and resource consuming.

Modular Update. In contrast to full image update, modular update preserves the application state as it does not require a reboot after code loading. SOS as one of the most popular approaches in this category was investigated in detail in the previous section. Contiki is an approach similar to SOS, providing a kernel on which Contiki-compliant modules can be dynamically reconfigured. Besides the fact that Contiki's linking mechanism is very memory consuming (due to creating large linking table), it fails to provide a concrete memory management mechanism for loading new modules. In [Koshy and Pandey 2005], a modular update system is proposed for the Mica2 mote to limit updates to the program functions. Each updated function, remotely linked at some sink node, is written at its original memory address, without needing to shift unchanged functions. In this approach, the sink node may need to keep per-node address maps and more importantly the update model suffers from lack of flexibility.

Reconfigurable Components. FlexCup [Marrón et al. 2006] attempts to bring reconfigurability to TinyOS, where TinyOS's component model, NesC [Gay et al. 2003], was not initially reconfigurable. This is achieved by sending the updated binary image of the component to the nodes and linking the new component to the system using a global symbol table. Despite the fact that FlexCup requires a reboot after a reconfiguration, the FlexCup metadata, including the application-wide symbol table and the relocation table of all components, imposes a significant

overhead to the external memory.

Coulson et al. in [Coulson 2008] propose OpenCom as a generic reconfigurable component model for building system software without dependency on any platform environment. The authors argue that they have tried to build OpenCom with negligible overhead for supporting features specific to a development area, however it is a generic model and basically developed for resource-rich platforms. Gridkit [Grace et al. 2006] is an OpenCom-based middleware for sensor networks, realizing coordinated distributed reconfigurations based on policies and context information provided by a context engine. This middleware was deployed on resource-rich Gumstix-based [GUMSTIX 2004] sensor nodes for a flood-monitoring scenario, where the minimum memory requirement of Gridkit core middleware and OpenCom run-time is about 104 KB. Lorien [Porter and Coulson 2009] is an OpenCom-driven approach that was recently proposed to provide a fully reconfigurable OS platform in WSNs. This work, however, is in initial stages of development and its performance has not been demonstrated.

FiGaRo [Mottola et al. 2008] is a WSN-specific reconfiguration solution, focusing on *what* should be reconfigured and *where*. The published work on FiGaRo fails to explain clearly its reconfiguration model and mostly focuses on a code distribution algorithm for WSNs. LooCI [Hughes et al. 2009] is a component-based approach, providing a loosely-coupled component infrastructure focusing on an event-based binding model for WSNs, while the Java-based implementation of LooCI limits its usage to the SunSPOT sensor node. Finally, Think [Fassino et al. 2002] is a C implementation of Fractal [Bruneton et al. 2006] whose main goal is to provide fine-grained reconfiguration at architecture-level. Think shares with RemoWare the capability to specify at compile time a subset of application's artefacts susceptible to be reconfigured [Loiret et al. 2009], even though it comes with a high overhead per reconfigurable component. It would be possible to implement more efficient optimizations within Think related to the way dynamic interactions are reified at run-time. However, these optimizations would lead to modifying the core implementation of the Think framework, a non-straightforward task.

Middleware Solutions. Impala [Liu and Martonosi 2003], designed for use in the ZebraNet project, is a middleware architecture that enables application modularity, adaptivity and reparability in WSNs. The main drawback of Impala is that updates are coarse-grained since cross-references between different modules are not possible. Costa et al. in [Costa 2007] propose RUNES to provide primary services needed in a typical resource-limited node. Specifically, their work supports customizable component-based middleware services that can be tailored for particular embedded systems. RUNES mainly focuses on Unix-based and Java-based platform, and its implementation on WSNs basically relies on the OS's dynamic facilities, *e.g.*, in [Costa 2007] Contiki's modular update model is exploited. Finally, Horr et al. propose DAVIM [Horré et al. 2008] as an adaptable middleware to allow dynamic service management and application isolation in WSNs. DAVIM considers the reconfiguration issue from the view of dynamic integration of services, whereas our work enables dynamicity within the structure of services.

8. CONCLUSION AND FUTURE WORK

To ensure the evolution of software along their lifespan, sensor nodes require to be dynamically reconfigurable. The middleware solution we presented in this paper, RemoWare, addresses this crucial issue by paying particular considerations to the resource and energy overheads induced by the reconfiguration process. RemoWare includes a set of optimized reconfiguration services deployed on the sensor nodes, which consistently update the required pieces of code. These services include binary update preparation, code distribution, run-time linking, dynamic memory allocation and loading, and system state preservation. This fine-grained reconfiguration process is achieved thanks to the adoption of a component-based approach in order to clearly isolate the boundaries of reconfigurable sensor software parts. The evaluations we performed on RemoWare illustrate that our solution implements a comprehensive reconfiguration process for WSN software with less overhead than comparable work with respect to memory and energy consumption. Our future work includes further deploying and testing of RemoWare in the home monitoring scenarios.

ACKNOWLEDGMENTS

This work was partly funded by the Research Council of Norway through the project Scalable Wireless Sensor Networks (SWISNET), grant number 176151.

REFERENCES

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. 2006. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practice and Experience 36,* 11-12, 1257–1284.

Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., and Luo, L. 2008. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, Raleigh, NC, USA, 85–98.

Cheng, J. and Kunz, T. 2009. A survey on smart home networking. *Technical Report SCE-09-10, Department of Systems and Computer Engineering, Carleton University*.

CORBA. 2006. Corba component model specifications. `http://www.omg.org/spec/CCM/4.0`.

Costa, P. e. a. 2007. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*. IEEE Computer Society, Washington, DC, USA, 69–78.

Coulson, G. e. a. 2008. A generic component model for building systems software. *ACM Trans. Comput. Syst. 26,* 1, 1–42.

Dunkels, A., Finne, N., Eriksson, J., and Voigt, T. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, Boulder, Colorado, USA, 15–28.

Dunkels, A., Osterlind, F., Tsiftes, N., and He, Z. 2007. Software-based on-line energy estimation for sensor nodes. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*. ACM, Cork, Ireland, 28–32.

F. Bachmann, L. e. a. 2000. *Component Software: Beyond Object-Oriented Programming*. Carnegie Mellon Software Engineering Institute.

Fassino, J.-P., Stefani, J.-B., Lawall, J. L., and Muller, G. 2002. Think: A software framework for component-based operating system kernels. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 73–86.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*. ACM, 1–11.

Gershenfeld, Raffi, N., Krikorian, R., and Cohen, D. 2004. The internet of things. *SCIENTIFIC AMERICAN*, 76–81.

Grace, P., Coulson, G., Blair, G., Porter, B., and Hughes, D. 2006. Dynamic reconfiguration in sensor middleware. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*. ACM, Melbourne, Australia, 1–6.

Gu, T., Pung, H. K., and Zhang, D. Q. 2004. Toward an osgi-based infrastructure for context-aware applications. *IEEE Pervasive Computing 3*, 66–74.

GUMSTIX. 2004. Gumstix embedded computing platform specifications. `http://www.gumstix.com`.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. 2005. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM, Seattle, Washington, 163–176.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. *SIGPLAN Not. 35,* 11, 93–104.

Horré, W., Michiels, S., Joosen, W., and Verbaeten, P. 2008. Davim: Adaptable middleware for sensor networks. *IEEE Distributed Systems Online 9,* 1, 1.

Huebscher, M. C. and McCann, J. A. 2004. Adaptive middleware for context-aware applications in smart-homes. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*. MPAC '04. ACM, Toronto, Ontario, Canada, 111–116.

Hughes, D., Thoelen, K., Horré, W., Matthys, N., Cid, J. D., Michiels, S., Huygens, C., and Joosen, W. 2009. Looci: a loosely-coupled component infrastructure for networked embedded systems. In *MoMM '09: Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*. ACM, Kuala Lumpur, Malaysia, 195–203.

Hui, J. W. and Culler, D. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, Baltimore, MD, USA, 81–94.

Jeong, J. and Culler, D. 2004. Incremental network programming for wireless sensors. In *SECON '04: Proceedings of the IEEE Sensor and Ad Hoc Communications and Networks*. Santa Clara, CA, 25–33.

Koshy, J. and Pandey, R. 2005. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *EWSN '05: Proceedings of the third European Conference on Wireless Sensor Networks*. Springer-Verlag, Istanbul, Turkey, 354–365.

Kulkarni, S. S. and Wang, L. 2005. Mnp: Multihop network reprogramming service for sensor networks. In *ICDCS '05: Proceedings of the Distributed Computing Systems, International Conference on*. IEEE Computer Society, Los Alamitos, CA, USA, 7–16.

Levis, P., Patel, N., Culler, D., and Shenker, S. 2004. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*. USENIX Association, San Francisco, California, 2–2.

Liu, T. and Martonosi, M. 2003. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, San Diego, California, USA, 107–118.

Loiret, F., Navas, J., Babau, J.-P., and Lobry, O. 2009. Component-based real-time operating system for embedded applications. In *CBSE '09: Proceedings of the 12th International Symposium on Component-Based Software Engineering*. Springer-Verlag, East Stroudsburg, PA, USA, 209–226.

Marrón, P. J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., and Rothermel, K. 2006. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *EWSN '06: Proceedings of the third European Conference on Wireless Sensor Networks*. Springer-Verlag, Zurich, Switzerland, 212–227.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. C. 2004. Composing adaptive software. *Computer 37,* 7, 56–64.

MICROSOFT COM. 1993. `www.microsoft.com/com`.

Milenković, A., Otto, C., and Jovanov, E. 2006. Wireless sensor networks for personal health monitoring: Issues and an implementation. *Computer Communications (Special issue: Wireless Sensor Networks: Performance, Reliability, Security, and Beyond) 29*, 2521–2533.

Mottola, L., Picco, G. P., and Sheikh, A. A. 2008. Figaro: fine-grained software reconfiguration for wireless sensor networks. In *EWSN'08: Proceedings of the 5th European conference on Wireless sensor networks*. Springer-Verlag, Bologna, Italy, 286–304.

Nehmer, J., Becker, M., Karshmer, A., and Lamm, R. 2006. Living assistance systems: an ambient intelligence approach. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*. ACM, Shanghai, China, 43–50.

OSOA. 2007. The service component architecture. `http://www.oasis-opencsa.org/sca`.

Pasztor, B., Mottola, L., Mascolo, C., Picco, G., Ellwood, S., and Macdonald, D. 2010. Selective reprogramming of mobile sensor networks through social community detection. In *EWSN '10: Proceedings of the 7th European Conference on Wireless Sensor Networks*. Lecture Notes in Computer Science, vol. 5970. Springer Berlin / Heidelberg, 178–193.

Porter, B. and Coulson, G. 2009. Lorien: a pure dynamic component-based operating system for wireless sensor networks. In *MidSens '09: Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. ACM, Urbana Champaign, Illinois, 7–12.

Ranganathan, A. and Campbell, R. H. 2003. A middleware for context-aware agents in ubiquitous computing environments. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag, Rio de Janeiro, Brazil, 143–161.

Reijers, N. and Langendoen, K. 2003. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. ACM, San Diego, CA, USA, 60–67.

Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Trung, Q. L., and Eliassen, F. 2010. Programming sensor networks using Remora component model. In *DCOSS '10: Proceedings of the 6th International Conference on Distributed Computing in Sensor Systems*. Springer-Verlag, Santa Barbara, CA, 45–62.

Taherkordi, A., Rouvoy, R., Le-Trung, Q., and Eliassen, F. 2008. A self-adaptive context processing framework for wireless sensor networks. In *MidSens '08: Proceedings of the 3rd international workshop on Middleware for sensor networks*. ACM, Leuven, Belgium, 7–12.

Tsiftes, N., Dunkels, A., He, Z., and Voigt, T. 2009. Enabling large-scale storage in sensor networks with the coffee file system. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. IEEE Computer Society, Washington, DC, USA, 349–360.

University of Oslo. 2010. The Remora Component Model. `http://folk.uio.no/amirhost/remora`.

Wang, Q., Zhu, Y., and Cheng, L. 2006. Reprogramming wireless sensor networks: challenges and approaches. *IEEE Network 20,* 3, 48–55.

Wood, A. e. a. 2006. Alarm-net: Wireless sensor networks for assisted-living and residential monitoring. *University of Virginia Computer Science Department Technical Report*.

Yang, J., Soffa, M. L., Selavo, L., and Whitehouse, K. 2007. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*. ACM, Sydney, Australia, 189–203.